DYNAMIC PARTITIONING OF SOCIAL NETWORKS

BY

MINDI YUAN

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering in the Graduate College of the University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Assistant Professor Yi Lu
In this thesis, I study the problem of dynamic partitioning of online social networks (OSN). The problem is practically important since it lays the foundation of many applications. If OSN users are treated as vertices and their connections, like friendships or conversations, as edges, social networks can be represented as graphs. Partitioning the OSN graph itself is difficult as its power-law degree distribution leads to many cross-partition edges. Moreover, unlike traditional graphs, which are more static, OSN graphs often evolve significantly due to dynamic changes in social interactions and hence the social network can be viewed as a stream of graphs sampled at different time. Therefore, it is desirable to partition the graphs not only in the spatial dimension, but also in the time dimension, which makes the problem more difficult. However, these evolutions, like the changing of conversation frequency between two friends, are not totally random. Thus if these evolutions can be captured, predicted, and used properly, they will help us achieve better partitioning. As a result, we propose to make use of past graph evolution information and encode them into edge weights. We develop two corresponding methods: (1) weight the edges based on access frequency, assuming users involved in frequent conversations are more likely to stay in the same partition; or (2) weight the edges based on access recency, assuming a graph consisting mostly of newest edges will reflect the current status of the network. We then design two separate algorithms based on these two definitions of edge weights. Simulations show that each method captures desired graph characteristics respectively and achieves dynamic partitioning.
To my parents, for their love and support.
Thanks to David Stein, Berenice Carrasco, Joana M. F. Trindade, and Yi Lu for their joint work in Chapter 1. Thanks to Alan Mislove of Rice University for providing the data on Facebook New Orleans network.

Thanks to Kun-lung Wu, Gabriela Jacques da Silva, and Yi Lu for their joint work in Chapter 2. The research in this chapter was sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under the Social Media in Strategic Communication (SMISC) program, Agreement Number W911NF-12-C-0028. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Defense Advanced Research Projects Agency or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. Mindi Yuan and Yi Lu are partially supported by NSF grant CNS-1150080.
# TABLE OF CONTENTS

CHAPTER 1  INTRODUCTION .................................................. 1

CHAPTER 2  PARTITIONING ACCESS-FREQUENCY WEIGHTED NETWORKS ........................................... 4
  2.1  Overview ............................................................. 4
  2.2  Related Work ...................................................... 9
  2.3  Activity Network and Facebook Wall Posts ....................... 10
  2.4  Two-Hop Neighborhood ......................................... 13
  2.5  Algorithm Design ................................................. 17
  2.6  Evaluation ......................................................... 20
  2.7  Summary .......................................................... 24

CHAPTER 3  PARTITIONING RECENCY WEIGHTED NETWORKS ......................................................... 25
  3.1  Overview .......................................................... 25
  3.2  Related Work ...................................................... 28
  3.3  Evolution-Aware Partitioning .................................... 30
  3.4  Evolution Analysis ............................................... 40
  3.5  Experiments ....................................................... 42
  3.6  Summary .......................................................... 54

CHAPTER 4  CONCLUSION ...................................................... 56

REFERENCES ................................................................. 58
In recent years, the world has witnessed an explosive growth in social media applications, both in terms of scale and dynamics. For example, Facebook has reached 955 millions registered users by June 2012 [1], and Twitter, with 500 million users by April 2012, generates over 140 million tweets every day and handles over 1.6 billion queries daily [2]. More importantly, the topics of interest on such social networks often change with time, sometimes rapidly, due to the endless interesting things happening in the world at different times. Because of its vast scale and fast dynamics, social media applications have created significant impacts on many aspects of human lives, including politics, business, product marketing, and so on. Organizations are forced to come up with strategies to cope with them. On the other hand, the OSN queries differ significantly from traditional web applications: The highly personalized content results in queries involving multiple small records generated by different users in the network [3, 4]; data is highly interconnected due to the community structure among users [5] whose popularity and structure vary over time [6]. As a result, there are tremendous interests in understanding the various properties of social graphs, how they evolve with time, and how the queries can be answered efficiently, preferably in real time as they are happening.

Graphs have been widely used to capture relationships between entities. Entities can be people, computer IP addresses, phone numbers, web pages, locations, molecules, and so on. Relationships can be social ties or interactions between people, communication links between computers, call records between phone numbers, links between web pages, road segments between locations, chemical bonds between molecules, and so on. These graphs contain valuable information about how the entities behave and interact among themselves. If properly captured, analyzed, and understood, such information can be very useful for many applications, such as online marketing,
Partitioning OSN graphs is practically important because it lays the foundation of many applications. For example, the entire social network is too big to be stored in one machine. Thus the graph needs to be divided into parts and stored in different machines. Due to interactions of OSN users, one often needs to retrieve another’s data, which could generate expensive inter-machine communications, if they are not stored in the same machine. Thus we need to consider how to partition the social network so that the communication cost is minimized. Namely, if possible, it is better to keep the users who will retrieve data from each other in one machine/partition. Most of the retrieval will then be local and the inter-machine communications will be reduced. Another example is the targeted recommendation system. In such a system, we are always interested in finding a special group of people to send them personalized advertisements. In this case, the graph needs to be partitioned according to user interests. Sometimes, just identifying the groups is not enough. We need to keep track of the real status of these groups, i.e. who its current members are, who left the group, and so on. For instance, if the system wants to broadcast a car ad to its potential customers, it is better to send them the ad before they buy a car. The users who already bought a car should be removed from the group. If they continue to receive these ads, which are now not useful to them, they may have a negative impression of the recommendation system. Therefore, a good system should be able to send the right ad to the right person at the right time.

OSNs are much more dynamic than traditional graphs. They could evolve significantly during a period of time. However, these evolutions are not totally random. For example, one pair of friends, who talked with each other actively yesterday, are more likely to keep in touch today than another pair of friends who did not really talk. In fact, we can make use of these evolutions to achieve better partitioning and hence their practical benefits, like the efficient distributed storage of social networks and the real-time targeted recommendation system, as mentioned previously.

In this thesis, we will use past graph evolution information and encode it into graph edge weights. We propose two methods targeting two types of systems: (1) weight the edges based on access frequency, assuming users involved in frequent conversations are more likely to stay in the same partition;
or (2) weight the edges based on access recency, assuming a graph consisting mostly of newest edges will reflect the current status of the network. We then develop corresponding algorithms based on these two definitions of edge weights. Simulations show that each method captures desired graph characteristics respectively and achieves dynamic partitioning.

The rest of the thesis is organized as follows. In Chapter 2, we present the method and simulation results for partitioning access frequency weighted graphs. In Chapter 3, we present those for partitioning recency weighted graphs. We offer conclusions in Chapter 4 and briefly discuss how to choose the edge weighting method for real applications.
CHAPTER 2
PARTITIONING ACCESS-FREQUENCY WEIGHTED NETWORKS

2.1 Overview

An example of OSN queries is the news feed of friends’ activities in the social network. This is a common feature of many OSNs, including Facebook and Twitter. The most recent messages posted in the friends’ network are retrieved and displayed in a page. More pages that include messages further in the past can be retrieved. For illustration purposes, we use the friends concept as that on Facebook and build a friendship graph where each user is a node, and a friendship relation is an edge. Figure 2.1 shows a friendship network of five users (squares). A user can post data that are visible to all his/her friends or post a message addressed to a particular friend, like a wall post in Facebook. The former data is stored at the user node, while the latter data is stored at the message nodes, which are placed between the sender and receiver of the message and indexed by the pair of sender and receiver IDs. As a user can observe all his/her friends’ activities, a message posted by or to a friend is also visible to the user. This constitutes a two-hop neighborhood for retrieval. For instance, user 1 can observe the message posted by user 2 on user 3’s wall as users 1 and 2 are friends. We focus on the inter-user messages in this chapter as they pose major challenges: For local queries, we need to keep the two-hop neighborhood in one partition, but a two-hop neighborhood can be much larger than its one-hop counterpart as the graph size grows exponentially.

Most OSN sites use hash-based partitioning of data for its simplicity, where the base unit of partitioning is a row of a table, hence also referred to as horizontal partitioning. Twitter, a micro-blogging social network with around 175 million users, developed Gizzard [7], which implements range partitioning. Cassandra [8] is a distributed storage system developed by Face-
book, which uses consistent hashing [9] of user IDs for better performance with incremental and dynamic data. Amazon implemented Dynamo [10] for storing and retrieving user shopping carts with a modified consistent hashing that addresses the non-uniformity of load distribution.

Hash-based partitioning techniques require the access of multiple small records located on different servers. It has been reported that distributed queries reduce performance compared to local queries [11, 12]. In particular, Curino et al. [11] reported that using local queries doubles the throughput. Retrieving small records from distributed servers also increases communication overhead as the payload of packets is small compared to the header.

However, local queries are difficult to achieve in OSNs due to the power-law degree distribution of the friendship graph [13] and the time-varying nature of queries and underlying social networks, which make most queries nonrepeatable.

1. **Power-law graph.** It is well known that balanced partitioning of power-law graphs is a difficult problem [14]. The friendship graph of social networks, where two users are connected if they are friends, has a power-law distribution and a small fraction of nodes has very large degrees. As a result, partitioning the graph into disjoint parts results in many queries accessing multiple partitions [15]. Replication approaches were proposed to keep a copy of the one-hop network for each node in the same partition. The recently proposed replication algorithm, SPAR [12], while significantly outperforming other algorithms in terms of replication overhead, has an average of seven extra replicas for the Facebook data with 512 servers, and a small portion of user data even needs to be replicated on all servers. Hence, replication of two-hop neighborhoods for each node will require a humongous amount of extra storage.

2. **Time-dependent queries and networks.** An example of time-dependent queries is to refresh of a user’s Facebook news feed to retrieve the most recent
activities of his/her friends: The set of *most recent* messages, by definition, varies with time, and depends on the user’s access frequency and the relative frequencies of message posting in the two-hop neighborhood. Such a query will rarely be repeated exactly after a relatively long time due to its time dependency. As a result, approaches that exploit past query patterns do not handle time-dependent queries well. In addition, the underlying social network also varies over time. Friends are added and deleted, and an existing friendship link varies in its interaction frequency, which makes partitioning solely based on friendship graphs insufficient.

2.1.1 Our Contribution

We propose to partition not only the spatial network consisting of user nodes, but also along the *time* dimension. That is, the partitioning will not place all messages between user 1 and 2 on the same server, but will divide these messages according to their time stamps and place messages within a particular time range on the same server. While the friendship graph considered by SPAR is power-law with a heavy tail, the graph corresponding to messages within a time range, say a month, has a much lighter tail. The phenomenon is observed in [6] and the latter graph is referred to as the *activity network*. As a result, partitioning activity networks produces balanced partitions with much fewer cross edges and reduces the need for replication. It also adapts to the time-varying nature of queries and networks. The idea is illustrated in Figure 2.2.

Figure 2.2 shows a small network with seven user nodes (squares). The messages exchanged between users *i* and *j* are indexed by (*i*, *j*) and represented by a message node (circles). In period 1, users 1 and 2 exchanged 100 messages while users 2 and 3 did not interact; In period 2, users 1 and 2 did not interact, but users 2 and 3 exchanged 100 messages. When replication is used to ensure one-hop locality on the friendship network as in [12], all 100 messages are replicated once as illustrated in Figure 2.2(b). However, when activity networks of different periods are partitioned, all messages are stored locally with no replication, as illustrated in Figure 2.2(c).

We use activities on the friendship graph to predict future query patterns and partition newly created messages accordingly. Our contributions are as
Figure 2.2: Example of a small social network to be partitioned on two servers. (a) Communication between users 1, 2, and 3 in the network over different periods of time. Number in square bracket is the total number of messages exchanged between the user pair. (b) Replicating one-hop network in the friendship graph. A total of 100 messages are replicated. (c) Partitioning the activity networks for different periods of time. All messages are stored locally.

follows:

• We analyze the pattern of wall posts in the New Orleans network on Facebook in Dec 2006. There are a total of 8640 active users and 13948 wall posts. We found that while most user pairs have only 1 post in the month considered, the activity is much more intense in the two-hop neighborhood, with the top 8% of users receiving more than 100 posts in its two-hop neighborhood. There is strong time correlation among wall posts between the same pair of users, and the network structure and activity intensity change significantly over a year.

• We construct an activity prediction graph (APG) that considers the two-hop neighborhoods such that a good partition on the APG cor-
responds to a large fraction of retrievals with local data. We show with simulation that partitioning on the APG produces superior data locality compared to partitioning on the original activity network.

- With static periodic partitioning of the APG using KMetis [16], data locality is improved 5.6 times over the hash-based partitioning when evaluated with queries of the most recent messages on Facebook traces.

- We propose a local partitioning algorithm that adapts to changes in activities while keeping movements across partitions small. Data locality is improved by 6.4 times over the hash-based partitioning when evaluated with queries of the most recent messages on Facebook traces.

Unlike full replication algorithms like SPAR [12], we do not guarantee that all users have their data locally. Instead, we aim to provide frequent users with high data locality, which speeds up a large proportion of queries with much less storage than algorithms with stronger locality guarantees. The problem is different from community detection [17, 18, 19] as the amount of data locality depends on the two-hop neighborhood: Even if two users do not interact frequently, hence not in the same community, one user can contribute a significant amount of activity in the other’s two-hop neighborhood. To exploit the strong time correlation, we also need to identify the change in activity at a much smaller timescale. The local algorithm solves a problem different from [20] as partitioning in our scenario is used for the placement of newly created messages. While balanced partitions are preferred, the resulting distribution of new messages across partitions is random, and cannot be fully controlled as with computation tasks [20]. The movement of message nodes also incurs a much smaller cost than [20] as no actual computation tasks need to be moved across servers. With the local algorithm, we only replicate the most recent few posts when moving a message node so that the next retrieval has all recent posts in the same partition.

For this chapter, we use data from the Facebook New Orleans network between Jan 2005 and Dec 2006 provided by Viswanath et al. While friendship data is publicly available, data on wall posts is more difficult to obtain due to privacy constraints. We did not use the data from Twitter and Orkut, which are also available to the public, because they do not contain time stamp information.
The rest of the chapter is organized as follows. We discuss related work in Section 2.2. Section 2.3 analyzes the activity patterns in the social network. Section 2.4 defines the activity prediction graph based on the two-hop neighborhood. Section 2.5 describes the static periodic algorithm and the local dynamic algorithm. Section 2.6 presents the evaluation results from emulation of Facebook page downloads. The conclusion is in Section 2.7.

2.2 Related Work

There has been significant interest in efficient data back-ends of online social networks recently. We compare and contrast our approach with related work in the area.

**Distributed Hashing:** Most online social networks rely on distributed hashing [8, 10] to partition data in a scalable way, but it can lead to poor performance due to lack of data locality, and create issues like the “multi-get” hole for Facebook. Our algorithm is shown to produce better data locality than distributed hashing in this chapter.

**Partitioning and Replication:** Pujol et al. proposed SPAR [12] to partition and replicate the graph to achieve one-hop locality. While outperforming existing algorithms, the average number of extra replicas is large, requiring several times more storage than simple hashing. This is due to the power-law nature of social networks and is difficult to overcome. The situation will be worse if two-hop neighborhoods are considered.

**Workload-Driven Approach:** Curino et al. proposed Schism that partitions data based on query patterns. It works well where queries are static and repeated many times. However, it will not be able to predict future queries in social networks, when both data and the network are changing over time.

**Community Detection Problem:** Nguyen et al. [17] presented an adaptive modularity-based method for identifying and tracing community structure of dynamic online social networks. Community structure is obtained based on the one-hop graph. Our problem requires the consideration of the two-hop neighborhood and exploits the strong time correlation in messages.
2.3 Activity Network and Facebook Wall Posts

While links in a friendship network represent the existence of relations, links in activity networks are weighted and represent the amount of activity associated with a relation. The latter links are added and removed more frequently as activities vary over time. Activity networks were shown to display very different structural properties than the friendship network [21, 22]. For instance, it was shown in [6] that the average degree of the activity network constructed from the New Orleans network on Facebook is much lower than that of the corresponding friendship network (only 12.2% of the social links showed any wall post activity), and that the maximal degree in the activity network is around 100. This motivates our algorithm to partition the activity network, as high data locality can potentially be achieved with few replicas.

In this section, we further explore the properties of the wall post activity in an OSN. In particular, we want to know the distribution of activities in the network and their correlation in time. We also explore the change in activity over time between a pair of users. This would enable us to make predictions for future activities.

2.3.1 Wall Post Distribution

First we plot the distribution of wall posts across message nodes, user nodes, and the two-hop neighborhood, which are defined formally in Section 2.4. In short, a two-hop neighborhood of a user includes the interactions with his/her neighbors and the interactions between the neighbors and their neighbors.

![Figure 2.3: Distribution of message node activities in Dec 2006.](image)
Figure 2.3 shows the distribution of the message node activities in the month of Dec 2006. About 94% of message nodes have only one event in the month, and all message nodes have at most five events. Figure 2.4 shows the distribution of the user node activities in Dec 2006. Sending a message counts as one event. While most of the users sent at most three messages, a significant portion of users sent more than 10 messages. The activity in a two-hop neighborhood is significantly more intense. Figure 2.5 shows the cumulative distribution of messages in a two-hop neighborhood for Dec 2006. The proportion of two-hop neighborhoods with at most 100 events is over 90%, which implies the number of messages exchanged in the two-hop neighborhoods is much larger. This makes the problem hard as there do not exist a few message nodes that are responsible for a large fraction of the interactions. In order to achieve fast retrievals, it is necessary to keep the large number of nodes in a two-hop neighborhood in the same partition.

Define the average auto-correlation function $\overline{R(i)}$ as follows. Take a user’s two-hop neighborhood, gather the messages $\{m(1), m(2), ..., m(k)\}$ in the two-hop neighborhood in the specified month. Compute the “auto-
correlation” function $R(i)$ of the sequence of messages:

$$R(i) = \frac{1}{k} \sum_{j=1}^{k} f(m(j), m(j + i)),$$

where $f(x, y) = 1$, if $x = y$. The function $R(i)$ gives the empirical measure given that a message between two users occurred at the $j$th position, another message between the same users occurs at the $(j + i)$th position. We compute the $R(i)$’s for all the two-hop neighborhoods in the month and obtain the average value $\bar{R}(i)$, as shown in Figure 2.6. The $y$-axis has a log scale. The plot shows a high correlation in time among messages between the same pair of users. It is more likely that messages between a pair of users occur consecutively or close in time. This motivates a local dynamic algorithm that adapts to the high time-correlation among messages.

### 2.3.2 Evolution of Activity Network

It was shown in [6] that the links in the activity network tend to come and go over time, and the strength of ties exhibits a decreasing trend as the link ages. In order to see how past queries should affect the partitions for the current period, we plot the proportion of links in Dec 2006 that are present in a past period in Figure 2.7.

Observe that the temporal correlation decreases significantly with the distance from the current period. This motivates us to weight an interaction by a decay factor so that interactions taking place in the remote past have less effect on the partitioning for the current period.

We observe that a significant portion of links in the current period have not appeared in the last two years. In order to examine the strength of new
links versus old links of various ages, we plot the proportion of messages that appeared in links that are common to both the past and current periods in Figure 2.8.

On one hand, there is limited information available from past interactions for the current period, as many links, hence interactions, are created among users who have never interacted. This necessitates an efficient algorithm component that handles new links. On the other hand, some strong links persist and are identified by the larger number of messages per link for old links. An efficient partitioning algorithm needs to place messages on these strong links appropriately.

\section*{2.4 Two-Hop Neighborhood}

Consider a graph $G$ on the vertex sets $U$ and $V$ where $U$ is the set of users in the social network and $V$ is the set of interactions, or messages, between a pair of users. The message vertex $v \in V$ always has degree 2 and connects to
the two user vertices that are involved in the interaction. We index a node \( v \) by the unordered pair of node indices it is connected to. Figure 2.9 shows an example of a graph with six user vertices and five interaction vertices.

![Graph with six user vertices and five interaction vertices.](image)

We define the neighborhood set of a user vertex and an interaction vertex.

**Definition 1** Let \( \mathcal{N}_u \) denote the neighborhood of user node \( u \), where
\[
\mathcal{N}_u = \{ u', (u, u') \text{ if } u < u', (u', u) \text{ if } u > u' \} \cup (u, u') \in V \text{ or } (u', u) \in V \}.
\]

**Definition 2** Let \( \mathcal{N}_{u,u'} \) denote the neighborhood of message node \((u, u')\), where
\[
\mathcal{N}_{u,u'} = \{ u, u' \}.
\]

The neighborhood of a user node includes all the user nodes sharing a common message node, and the common message nodes. The neighborhood of a message node includes the two user nodes connected to it. Now we define the two-hop neighborhood of a user node and a message node.

**Definition 3** Let \( \mathcal{H}_u \) denote the two-hop neighborhood of user node \( u \), where
\[
\mathcal{H}_u = \bigcup_{u' \in \mathcal{N}_u} \mathcal{N}_{u'}.
\]

**Definition 4** Let \( \mathcal{H}_{u,u'} \) denote the two-hop neighborhood of message node \((u, u')\), where
\[
\mathcal{H}_{u,u'} = \mathcal{N}_u \bigcup \mathcal{N}_{u'}.
\]

A two-hop neighborhood of user \( i \) includes vertices whose content is visible to user \( i \). A user can view the interactions between the user and the user’s friends, and all interactions initiated or received by all the friends.
The messages initiated or received directly by user $i$ are on the one-hop neighborhood centered at user $i$, while the messages initiated or received by all friends of user $i$ reside on the two-hop neighborhood centered at user $i$. For instance, the set of user vertices $\{2, 3, 4, 5, 6\}$ and the edges connecting them constitute the two-hop neighborhood centered at user 1. The two-hop neighborhood centered at a message vertex is the union of the one-hop neighborhood of its initiators and receivers. For instance, the set of user vertices $\{1, 2, 3, 4, 5\}$ and the edges connecting them constitutes the two-hop neighborhood centered at message vertex $(1, 2)$. It is important that we consider two-hop neighborhoods as each message can be accessed by any user on its two-hop neighborhoods.

Consider the following retrieval scenario. Each user is assigned an access frequency $m_i$ and retrieves the data in the two-hop neighborhood. Each message node stores messages and is assigned a weight $w_{i,j}$. Potential values for $w_{i,j}$ are the number of messages at the node, or a weighted sum of the messages. The objective is to define weights $e^i_{i,j}$ and $e^j_{i,j}$, for the edges connecting user nodes $i$ and $j$ to the message node $(i, j)$, so that minimizing the cross-partition edges in the graph will corresponds to maximizing locality for accesses.

**Definition 5** Let $D_i$ denote the sum of all message node weights in the two-hop neighborhood,

$$D_i = \sum_{(i,j) \in H_i} w_{i,j}.$$  

Let $W^k_{(i,j),i}$ denote the message node weights in a remote partition if the edge between user vertex $i$ and edge vertex $(i, j)$ is cut. Let $G_{\sim(i,j),i}$ denote the graph with the edge between $(i, j)$ and $i$ removed, and let $N_{\sim(i,j),i}$ and $H_{\sim(i,j),i}$ be the neighborhoods in $G_{\sim(i,j),i}$, then

$$W^k_{(i,j),i} = (H_k - H_{\sim(i,j),i}) \cup (i, j) \quad \text{if} \quad i \in N_k$$  

$$= (H_k - H_{\sim(i,j),i}) \cup i \quad \text{if} \quad (i, j) \in N_k.$$  

For instance, the total weight of messages accessible to user 1 in Figure
Figure 2.10: Graph with six user vertices and five message vertices, with weights on each message vertex.

2.10 is,

\[ D_1 = w_{3,6} + w_{1,3} + w_{1,2} + w_{2,4} + w_{2,5} \]
\[ = 22 + 46 + 71 + 5 + 10 = 154, \]

and the message weights in a remote partition for node 1 if the edge between (1, 2) and 1 is cut is

\[ W_{(1,2),1} = w_{1,2} + w_{2,4} + w_{2,5} \]
\[ = 71 + 5 + 10 = 86 \]

and the message weights in a remote partition for node 3 if the edge between (1, 2) and 1 is cut is

\[ W_{(1,2),3} = w_{1,2} = 71. \]

We are ready to define the edge weights.

**Definition 6** Let \( e_{i,j}^i \) denote the weight on edge between nodes \( i \) and \((i, j)\), then

\[ e_{i,j}^i = \sum_{k \in \mathcal{H}_{i,j}} m_k \frac{W_{(i,j),i}^k}{D_k}, \]

which is the sum of fraction of remote message weights weighted by user frequency.

Consider accesses at each user node as an independent Poisson process with intensity \( m_i \), and interaction between user \( i \) and \( j \) an independent Poisson process with intensity \( w_{i,j} \).

**Thm 1** For a tree graph, when user \( k \) accesses the OSN to retrieve a message in the two-hop neighborhood, the expected intensity of accesses that retrieve from a remote partition when the edge between nodes \( i \) and \((i, j)\) is cut is \( e_{i,j}^i \).
For a graph with cycles, \( e_{i,j} \) is a lower bound for the expected number of messages from a remote partition when the edge between nodes \( i \) and \((i, j)\) is cut.

For a tree graph, when the edge between nodes \( i \) and \((i, j)\) is cut, the sum rate of messages in the remote partition is \( W_k^{(i,j),i} \), and the sum rate of all messages in the two-hop neighborhood is \( D_k \). Since all interactions are independent Poisson processes, the probability that a message is from the remote partition is \( \frac{W_k^{(i,j),i}}{D_k} \). Since the accesses of one user are Poisson processes independent of interactions, the probability that the most recent message at an access is in the remote partition is also \( \frac{W_k^{(i,j),i}}{D_k} \). The accesses of user \( k \) that retrieve a message from the remote partition form a thinned Poisson process with intensity \( m_k \frac{W_k^{(i,j),i}}{D_k} \). Consider all the users in the two-hop neighborhood of the node \((i, j)\), the aggregated process has intensity \( \sum_{k \in N_{i,j}} m_k \frac{W_k^{(i,j),i}}{D_k} \).

For a graph with cycles, the numerator of \( W_k^{(i,j),i} \) includes only the immediate neighbor of the node \((i, j)\). The actual partition can place more nodes in the remote partition, hence \( e_{i,j}^{(i,j)} \) provides a lower bound.

### 2.5 Algorithm Design

In this section, we define the static periodic partitioning algorithm and the adaptive local algorithm. We also discuss the implementation of the retrieval algorithms.

#### 2.5.1 Periodic Algorithm

The periodic algorithm computes a graph partition every month using the activity prediction graph (APG) obtained based on frequency of previous interactions. We define the activity prediction graph (APG) and computation of the interaction node weights \( w_{i,j} \) to be the discounted message frequency,

\[
w_{i,j} = C \sum_{k=1}^{K} f_k \ n_{i,j}^k,
\]
where $K$ is the total number of past periods considered, $C$ is a scaling constant, $n_{k,i,j}^k$ is the number of messages exchanged between users $i$ and $j$ in month $k$, and $f_k$ is the decay factor computed on a monthly basis for month $k$ defined as follows. Let $L_k$ be the set of links in month $k$ and $L$ be the set of links in the current month. Let $|\cdot|$ denote cardinality of a set, then

$$f_k = \frac{|L_k \cap L|}{|L_k|}.$$

The weight of each message vertex is used for computing balanced partitions as it is a prediction of the number of messages at each link based on past queries. We use KMetis, a software program from the METIS library [16], to partition the APG. KMetis uses a multilevel $k$-way min-cut algorithm to produce partitions that balance vertex weights in each partition and minimize edge weights across partitions.

For a message whose corresponding node is present in the APG, it is stored in the given partition. For a message not predicted by the APG, we use the following simple algorithm:

1. If both the initiator and receiver of the message exist in the APG, but no previous message exists, store the message with the user with a smaller value of $D$, as the new message will contribute a larger fraction of this user’s future query.

2. If exactly one of the initiator and receiver of the message exists in the APG, say user node $i$, store the new message in the same partition as node $i$.

3. If neither the initiator nor the receiver exists in the APG, store the new message to the partition with the least number of messages.

The values $D_i$ are updated for each user $i$ as new messages are stored in each partition.

2.5.2 Adaptive Local Algorithm

The periodic algorithm has two drawbacks: (1) It changes the placement of a large number of message nodes at the end of a period, creating artificial remote accesses for subsequent retrievals as two messages on the same message
node can reside in different partitions. (2) It fails to take advantage of the strong time correlation of messages as no repartitioning takes place within the period. This motivates the design of a local adaptive algorithm.

We propose a local algorithm that is triggered when a retrieval results in remote accesses. We define the boundary pairs

$$B = \{ (i, (i, j)) : i \text{ and } (i, j) \text{ are in different partitions} \}.$$ 

Only the subset of boundary pairs for which the weights in the two-hop neighborhood have changed since the last repartitioning will be considered. Changes in message weights outside the two-hop neighborhood can have an effect on the boundary pair, but are ignored to reduce complexity as the effect is usually small. We recompute the edge weights in the APG updated with current messages. For each pair in the boundary set, we consider the following reward function

$$F = -\Delta E - M,$$

where $\Delta E$ is defined as the change in cross-boundary cost in APG if the node in consideration is moved and $M$ is a parameter designating the cost of movement for one node. Movement occurs only when $F > 0$. For a message node $(i, k)$ currently in the same partition as $i$,

$$\Delta E = e_{i, k}^i - e_{i, k}^k$$

is used to decide whether to move it to the same partition as $k$. For a user node $i$, it can connect to multiple message nodes in different partitions, and

$$\Delta E = \max_{P \in \{\text{adjacent partitions}\}} \sum_{(i, k) \in P} e_{i, k}^i - \sum_{(i, j) \text{ local}} e_{i, j}^i.$$

To avoid the problem of having messages on the same message node in different partitions immediately after movement, we replicate the most recent few messages in the new partition. The discontinuity caused by movements is less of a problem with the local algorithm than with the periodic, as movements are distributed across time, with only a few movements daily. The new nodes are added in the same way as described in Section 2.6.1.
2.5.3 Implementation

When a user initiates a retrieval, its two-hop neighborhood is looked up from a hash table, and messages are retrieved accordingly. With the time-dependent partitioning algorithms, the same message node can reside in different partitions at different times. This requires an extra hash table lookup, which returns the timestamp at which a node changes its physical location, together with the number of messages from that time interval. The actual locations to be accessed by the retrieval are determined from the timestamps and number of messages.

2.6 Evaluation

We test the algorithms with messages in the month of Dec 2006. Each user is assigned an access process that retrieves the most recent six messages in the two-hop neighborhood. We choose the number of messages to be six as our data set is relatively small: the data in Dec 2006 contains a total of 13948 messages with 8640 active users. We demonstrate the advantage of considering the two-hop neighborhoods over the original activity network, and evaluate the performance of the periodic and local algorithms.

2.6.1 Periodic Algorithm

We compare our algorithm to two hash-based horizontal partitioning algorithms. These are the algorithms used in commercial online social networks. The first algorithm, hash\_p1, hashes the initiator ID of a message. As a result, all messages generated by the same user are grouped in one partition. The second algorithm, hash\_p1p2, hashes the unordered sender-receiver pair of a message. All messages exchanged between a particular pair of users are grouped in one partition. We compare the experiments with different number of partitions up to 20. We did not experiment with a larger number of partitions as there are only 8640 active users for Dec 2006, and we are considering the locality of messages in a two-hop neighborhood. We also show the results from a retrospective algorithm, denoted by “retro,” where the actual messages in Dec 2006 are included in computing the APG. This
is the optimal result for a static partitioning algorithm.

We use $C = 12$ and $K = 23$ to construct the APG. We experimented with other values of $C$ and $K$ and the result is not sensitive to the change in the values. We associate a frequency $m_i$ to each user $i$. For this experiment, we let

$$m_i = \sum_{(i,j) \in G} w_{i,j},$$

that is, the sum of all weights on message vertices connected to user vertex $i$. This assumes that the frequency of reads are proportional to the number of messages sent or received by a user. We do not consider the balance of accesses across partitions in this chapter, but it can be readily integrated by assigning weights to user vertices proportional to its frequency.

Figure 2.11: Proportion of queries that access only one partition. Comparison of the periodic algorithm with hashing the initiator ID (hash$_{p1}$) and the unordered initiator-receiver pair (hash$_{p1p2}$).

Figure 2.11 compares the proportion of queries that have all six most recent messages in a single partition for the three algorithms. With five partitions, the periodic algorithm produces 50% of all queries with all six messages in one partition whereas both hashing algorithms have less than 10% of local queries. With 20 partitions, the periodic algorithm achieves 34% of local queries as some two-hop neighborhoods need to be cut to keep the balance of the data storage, which is still over 12 times better than the hashing algorithms, each achieving 2.8% and 2.6% of local queries. In all cases, the performance of the periodic algorithm is within 80% of the retrospective algorithm, showing a good prediction quality of the APG.

Figure 2.12 compares the proportion of queries that have all six most recent messages in at most three partition. For all numbers of partitions, more than 90% of queries access at most three partitions with the periodic algorithm.
For the hashing algorithm, while 71% of all queries access at most three partitions when there are a total of five partitions, the fraction decreases to 40% when there are a total of 20 partitions. The performance of the periodic algorithm is more than 95% of that of the retrospective algorithm.

2.6.2 Two-Hop Neighborhoods

The edge weights defined in Theorem 1 reflect the access frequencies and message distribution in the two-hop neighborhoods. We highlight the advantage of using the edge weights with the following experiment.

We sample the access frequencies $m_i$ from a power-law distribution, where $\text{prob}(m_i > x) = x^{-1.1}$. We define the edge weights to be the number of messages on the original activity network, plus the access frequencies to account for the difference in read activities: $e_{i,j} = w_{i,j} + C \cdot m_i$. Recall that $C$ is the scaling constant set to 12. We refer to this definition of edge weights as one-hop. For each user $i$, $m_i$ read operations are randomly inserted into the trace of Dec 2006. This models the fact that more users read than write, and the read frequency can be large.

Figure 2.13 shows the performance comparison of the two-hop and one-hop edge weights. When the number of partitions is five, the two-hop edge weights achieve 17% more local queries than the one-hop. With 20 partitions, the two-hop is 37% better. We also compared the performance with other distributions of access frequency and found that the two-hop edge weights always perform as well as the one-hop, and significantly outperforms the latter in some cases.
Figure 2.13: One-hop vs. two-hop. Sample 11 random cases and average.

2.6.3 Local Algorithm

Figures 2.14 and 2.15 show the performance of the local algorithm with \( M = 10 \). With five partitions, the local algorithm results in 20% more local queries than the periodic algorithm and almost six times more than the hash algorithms. With 20 partitions, the local algorithm achieves 30% better than the periodic algorithm and 13 times better than the hash. Both the local and periodic algorithms have more than 90% queries accessing at most three partitions, with the local algorithm performing slightly better. The total number of movements for the local algorithm with five partitions is 1122, evenly distributed across time. This amounts to 40 movements daily, which is small. The number of movements increases with the number of partitions, reaching 1859 at 20 partitions.

Figure 2.14: Locality tests for the periodic algorithm, greedy \((M = 10)\) and hash_p1p2.

The static algorithm uses KMetis to produce balanced partitions of the APG. However, as APG only provides a prediction of the actual messages, the resulting evenness of message distribution is a random variable. The local algorithm only considers evenness by assigning new nodes to partitions with fewer messages. Figure 2.16 compares the evenness of distribution of
the messages in Dec 2006, defined by $\sum_{k=1}^{K} |W_k - W/K|/W$, where $W_k$ is the total message weights in partition $k$ and $W = \sum_{k=1}^{K} W_k$. There is no observable difference in evenness of actual messages between the local and periodic algorithms.

2.7 Summary

We proposed a partitioning method based on two-hop neighborhoods to improve the locality of retrievals in OSN. We proposed a static periodic algorithm using KMetis and a dynamic local algorithm that adapts to changes in the network. The local algorithm requires more computation, but produces better locality. It also controls the movements across partitions and distributes them across time. Both algorithms significantly improved data locality than hash-based partitioning.
CHAPTER 3
PARTITIONING RECENTY WEIGHTED NETWORKS

3.1 Overview

In Chapter 2, we studied the problem of partitioning access frequency weighted graphs. We have seen the past information of the graph will help us achieve better partitioning and the corresponding practical benefits: more local queries and less remote retrieval of data. But some other applications may care more about the current status of the graph. Thus, in this chapter, we study the problem of partitioning recency weighted graphs. According to the problem model in this chapter and for the convenience of representation, we treat the graph as a streaming graph where updates to the graph are given as a stream of edge or vertices insertions or deletions. A streaming graph is an incremental graph, so the entire structure is not given beforehand.

Consider an undirected weighted graph \( G = (V, E) \). Each edge is in the form of \( < v_i, v_j, w_{ij} > \), where \( v_i \) and \( v_j \) are the two endpoints of an edge and \( w_{ij} \) the associated weight. Usually a constraint is imposed in graph partitioning, either the total number of partitions or the maximum size of a partition. In this chapter, we assume that there is a constraint on the maximum number of vertices in each partition, i.e. the maximum partition size (MPS), denoted by \( M \). We want to partition the vertices \( V \) into partitions \( C_1, C_2, ..., C_n \) so that the sum weight of the interpartition edges (or called cut size) is minimized. However, unlike traditional partitioning, where the vertices can be moved around, we would delete edges while MPS constraint violation occurs. And if all the adjacent edges of a vertex are deleted, this vertex is removed from the graph automatically. An edge can also be deleted when it expires, caused by a sliding window. Windowing is defined in more detail in Section 3.2.1. The objective of evolution-aware partitioning is to capture the emerging partitions. And later we will see, all
the deleted edges are the oldest ones in the current partition or graph. So we prefer deletion, instead of moving vertices around. And we will also see later, if the edge weight is properly defined, the weighted cut size can reflect how well an algorithm captures the evolution. That is why we want to partition the graph to minimize the cut size.

The definition presented is for partitioning an existing graph. In a streaming graph, it is impossible to obtain the entire graph $G$ beforehand. Instead, a sliding or tumbling window is typically used to maintain a graph with only the most recent updates, such as the graph consisting of the last 1 million updates or the graph containing updates from the last one day or one week. As new edge updates continue to stream in, old updates are removed from the window for further partitioning consideration. In a streaming graph, $G(t)$ can be considered as a snapshot of the maintained graph within the window taken at time $t$ of a continuously evolving graph. Hence, we need to solve the presented partitioning problem whenever the graph is changed. This can happen when an edge is either added or removed. As a large volume of graph updates can occur in a short period of time, most of the offline graph partitioning algorithms, such as [17, 18, 23, 24], are generally too inefficient since they need to re-partition from scratch each time the graph is changed. As a result, an online incremental algorithm is preferred for partitioning streaming graphs.

Due to dynamic changes in social interactions, partitioning often evolves with time in streaming graphs. Even though a sliding window in streaming naturally captures some partition evolution, it alone might not be adequate. This is particularly true if the window size is large and the partitioning within it already evolves a lot. Prior graph partitioning methods, both offline and online, are mostly insensitive to the evolution [17, 18, 23, 24, 25]. Ignoring partitioning evolution has two major issues. (1) Although the partitions have changed into a different shape, an evolution-insensitive partitioning algorithm may still use an out-of-date version of the partition structures for partitioning, lowering the partitioning quality. (2) An unsuitable assignment of a vertex to a partition or an inappropriate splitting of a large partition into smaller subpartitions may cause more expensive operations in the future, causing unnecessary merging and splitting of partitions and reducing the efficiency of the algorithm.

In order to deal with this interesting yet challenging problem, we pro-
pose a new evolution-aware partitioning (EAP) approach to partitioning streaming graphs. We treat each connected component as a partition and maintain these partitions incrementally upon a graph update. We keep all the edges in each partition in a sorted order based on its recency. We favor more recent edges than older ones in our partition merging and splitting, effectively capturing emerging partition evolution in the streaming graph. The maintenance of recency-ordered edges within a partition makes our algorithm incremental and efficient. The insertion of an edge is mostly an $O(1)$ operation if there is no violation in the partition size constraint after an insertion. The deletion of an edge is always an $O(M)$ operation, where $M$ is the size of its residing partition. Additionally, our algorithm is amenable to distributed implementation. The maintenance of partitions can be distributed to different compute nodes and it will incur at most two compute nodes in merging two partitions.

We evaluate the effectiveness and efficiency of our evolution-aware partitioning approach against a previous state-of-the-art evolution-insensitive partitioning (EIP) approach [25] using both synthetic data and real-world graph data. Moreover, we propose a measure to quantify the stableness of partitioning in the graphs and show that in general the more unstable the graphs, the more advantageous EAP is.

Our contributions are as follows:

1. We propose an evolution-aware partitioning algorithm for streaming graphs. The algorithm quickly and accurately capture the evolution of partitions in the graph stream.

2. We design an evolution analysis scheme to quantify the stableness of an underlying stream. We compare the performance of EAP under streaming graphs of different stableness and show its advantages.

3. We conduct quality and throughput experiments to compare our algorithm with EIP and METIS using synthetic data sets and real-world data sets. The results show that EAP outperforms EIP in better partitioning quality (up to 6 times) and better throughput (up to 8 times).

The rest of the chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 describes the evolution-aware algorithm. Section 3.4
presents the evolution analysis scheme. We show the experiments in Section 3.5 and conclude in Section 3.6.

3.2 Related Work

Most of the graph partitioning algorithms are offline [17, 18, 23, 24]. One popular offline algorithm is METIS [24]. It is based on the multilevel graph partitioning paradigm [26], consisting of three phases: graph coarsening, initial partitioning, and uncoarsening. METIS is shown to produce high-quality (balanced) partitions. Yuan et al. [27] proposed a partitioning algorithm for fast retrieval of time-dependent queries in social networks. They first used METIS to partition a base graph. After that, upon each edge insertion, they used an online algorithm to determine whether the related vertices should be moved or not, in order to reduce the cut size. But they did not allow edge deletion, so it cannot be directly applied to a fully dynamic setting. In [28], Kernighan and Lin proposed an efficient heuristic for partitioning vertices of a weighted graph into subsets of given sizes such that the cut size is minimized. More offline algorithms can be found in this survey [14]. The major shortcoming of an offline algorithm is that it cannot efficiently deal with the large number of updates existing in streaming graphs.

For online streaming graphs, Aggarwal et al. [29] proposed an algorithm for clustering graph streams. They used a hash-based compression of the edges to create micro-clusters onto a smaller domain space. They showed that their method provides bounded accuracy in terms of distance computations. However, their algorithm does not deal with edge deletions, making it not applicable to partitioning streaming graphs in the face of sliding windows. Stanton and Kliot [30] designed a series of natural and simple heuristics for clustering large streaming graphs into distributed compute nodes. They showed that the heuristics are a significant improvement on a large collection of graph data sets, and the algorithms are scalable in the size of the graphs and the number of clusters. However, they assumed that the graph is already stored on disks, so they are able to stream the graph from disks in a certain order, such as DFS (depth first search) or BFS (breadth first search). In a general streaming setting, the orders of edges are not predictable.
Eldawy et al. investigated the problem of clustering streaming graphs, similar to the problem we study in this chapter. They developed a reservoir sampling based algorithm for achieving both good quality and high throughput. However, their scheme is not sensitive to cluster evolution. In addition, their scheme maintains the edges in a sorted order based on the random numbers assigned to them, making the clustering algorithm less efficient.

In terms of evolution analysis of streams and evolutionary partitioning, Aggarwal [31] et al. designed a framework for clustering evolving data streams. The data streams in [31] are not graph streams and the clustering objectives are different. Moreover, they did not quantify the evolution analysis, nor did they make use of evolution for better clustering. In contrast, we focus on making the partitioning algorithm aware of the cluster evolution to improve the quality of clustering in streaming graphs. Gupta et al. [32] proposed an evolutionary clustering method to analyze bibliographic networks. They clustered a sequence of snapshots of a graph and try to diagnose the evolution by tracking and comparing the clustering results. The algorithm for clustering each graph snapshot is still offline. They did not support fully dynamic maintenance of clusters upon edge addition or deletion, even though they did use previous clustering results to initialize the clusters for the next snapshot. In this chapter, we focus on continually capturing the newly emerging clusters.

For capturing emerging events, one can identify the densest subgraph(s) of a graph. Bahmani et al. [33] developed an algorithm to find locally dense components of a graph under a streaming model. Their model assumes that all the vertices of the graph are known and that edges arrive one by one. In [34], Angel et al. proposed a novel algorithm to maintain dense subgraphs under streaming edge weight updates for real-time story identification. This work also assumes that the graph is complete, i.e. every node connects to every other node. Neither of these works consider windowing scenarios. Agarwal et al. [35] designed an algorithm for real-time discovery of dense clusters in highly dynamic graphs by exploiting a novel graph property called short-cycle property. Although their streaming model is almost the same as ours, their definition of clusters is different. They are still finding the densest subgraphs. The problem of clustering is different from just finding the densest subgraphs. In this chapter, we partition the vertices of the graph into different partitions so that we can identify and follow the partition evolution.
This can be achieved by minimizing the weighted cut size of the graph while satisfying the maximum partition size constraint. But the densest subgraph problem aims at identifying a few densest partitions above a specified density threshold, rather than partitioning the entire graph.

3.3 Evolution-Aware Partitioning

In this section, we describe the details of our evolution-aware partitioning algorithm for streaming graphs. We first provide the intuition of this algorithm and then describe the system design and algorithm details. We also provide a possible distributed implementation of the algorithm.

3.3.1 Intuition

Let us first consider the following example in Figure 3.1. Suppose there is a soccer discussion forum during the European Championship 2012. The vertices $A, B, ..., G$ are seven users. Assume $A, B,$ and $D$ are fans of Germany, $C$ is a fan of Italy, and $E, F,$ and $G$ are fans of Spain. Suppose it is Jun 28 when there is a game between Germany and Italy and the winner will enter the final to play Spain. Assume that an edge represents an interaction between two users, such as a conversation or a reply to a message posted on the forum. New edges arrive in the order according to the number associated with it. So the first conversation happens between $B$ and $C$, i.e. the edge $<B, C, 1>$ comes in first. They are debating how one team will defeat the other. Then the edge $<A, C, 2>$ comes in. Followed is the edge $<E, F, 3>$ when the Spanish cannot wait to see the final and so on. After that day, unfortunately Germany lost. As a result, $A, B,$ and $D$ might not be so active as before. $C$, on the contrary, is so excited about the Italian team entering the final and this fan begins to talk to the Spanish fans, that is edge $<C, E, 10>$ and so on. Before time 10, there are naturally two separate partitions (the two clouds shown in Figure 3.1). But after that, $C$ is trying to talk to the members of the right partition (the dotted lines shown in Figure 3.1). The fact that $C$ now prefers talking to $E, F,$ and $G$, rather than $A, B,$ and $D$ as before, is an example of partitioning evolution.
Figure 3.1: “Where to cut?” The number associated with the edge is the timestamp/weight of that edge. Suppose the maximum partition size is 4. Each connected component is a partition.

What if we want to partition this example now? Suppose that the maximum number of vertices in one partition is four. Once there is an edge between $C$ and $E$, the two partitions will merge together. After that, there is only one big partition of size seven, so we have to split the overcrowded partition. Where shall we cut the graph? If the edges are of same weight, say each edge weighs 1 unit, we should remove the edge that just came in, that is, the edge between $C$ and $E$. This way the cut size is 1, which is the best. Similarly, when the 11th and 12th edges come, it is still best to cut the graph by removing the newest edges in this setting. However, the evolution here is clearly that $C$ wants to move to the partition on the right. But $C$ will continue to stay with the left partition after the 12th edge update. This means, we are not able to detect and adapt to the evolution that $C$ is now eager to talk to the partition containing $E$, $F$, and $G$, rather than the partition containing $A$, $B$, and $D$.

A better way is that we move $C$ to the partition on the right and allow them to form a new group/partition, i.e. splitting the partition by cutting the edges with timestamps 1, 2, and 4, which would result in a total cut size of 3 (removing 3 edges). This cost is always bigger than just removing the newest edge at each step when the MPS violation occurs (time 10, 11, and 12), so any smart algorithm will not do that. Note this statement is only true if an algorithm will “discard” edges. For example, METIS will never delete an edge. At time 12, it still remembers edges 10 and 11, so it may succeed in moving $C$ to the right. That is why we will see later METIS can also capture the evolution very well. The only problem is its inefficiency, which is crucial for an online application.

How can we allow the algorithm to automatically follow the partitioning
evolution as it happens? Our intuition is to use recency. We would take the
recency into consideration in our partitioning algorithm and put more weight
on a new edge than an old edge. For example, if we just use the ordering
of arrivals, or the timestamp of the edge, as the edge weight, then when the
10th edge comes in, we will cut edges 1, 2, and 4, which has a total weighted
cut size of seven and that would be the minimum. In this way, the weighted
cut size actually can reflect how well an algorithm can capture the evolution.

Even more importantly, keeping the newer edge and deleting the older one
for partitioning can also improve the partitioning efficiency. Suppose it is still
an unweighted graph in Figure 3.1, then we will discard the 10th edge upon
its arrival. This will result in discarding more edges in the following steps:
the 11th and 12th edges, too. Actually all these discarding steps are solving a
violation of the MPS constraint. Later in the chapter we will see that solving
the violation of MPS is generally expensive for a partitioning algorithm,
which can significantly degrade its efficiency. As a result, we aim to avoid
the violations as much as possible, if we can, during the execution of the
algorithm. Fortunately, once we label each edge with a weight proportional
to its timestamp, the number of violations will be smaller. For example, for
the weighted graph in Figure 3.1, after the arrival of the 10th edge, we will
remove the first four edges for solving the violation. When the 11th and 12th
edges come, there will be no violations anymore. In summary, if using an
unweighted graph, there will be three violations in total for inserting these
12 edges; but if using a weighted graph, there will be only one violation
during the whole process. Hence if an algorithm can follow the evolution
trend better, then there tends to be fewer unnecessary violations in total.
This will also improve the algorithm efficiency.

As a result, the idea of the evolution-aware partitioning algorithm is quite
simple: upon a graph update, always keep the newest edge and remove the
oldest edges first if a violation occurs on the constraint of the maximum parti-
tion size. This algorithm not only effectively captures emerging partitioning
evolution but also reduces the number of constraint violations, increasing
partitioning efficiency.
3.3.2 System Design

Figure 3.2 is an overview of the system architecture. The first building block is called window manager (WM), and the second one is graph manager (GM). Once a new graph update comes, it will enter the system through the WM. The WM will then forward graph insertions or deletions to the GM. For a graph query, it will directly go to the GM and the GM is responsible for answering it.

![Figure 3.2: The system architecture.](image)

Window Manager

Windowing is generally used in the streaming environment to limit the amount of data for processing. For instance, most of the time, we care more about the most recent state of the graph, like the graph within the past 24 hours. Then we only need to keep those graph updates and evict the expired ones. In this sense, the streaming window naturally helps us capture some evolution, because the edges kept are always the newer ones and no old edges can stay in the window forever. But as mentioned before, it may not be adequate to capture all the partition evolution, especially when the window size is large and the partitioning within it evolves a lot.

There are generally two kinds of windows in streaming: sliding window and tumbling window. Both can be time-based or count-based windows. In this chapter, we will use count-based sliding window and count-based tumbling window. A count-based sliding window will maintain a specified window size, like the newest 1M updates, and evict the oldest one when a new update comes in and the window is full. A count-based tumbling window will store the new updates until it reaches the maximum window size, process them, and then flush all the stored updates out of the window. The tumbling window can be imagined as a “jumping” window. While the sliding window
moves one by one, the tumbling window will jump to a position right in front of its previous window so that they are next to each other, but not overlapped.

Note that it is also easy to optionally apply sampling before the window manager, as in [25]. Sampling can be used to sparsify the graph. It has been shown important and useful, especially when the graph is dense [36], for improving the efficiency of a graph processing algorithm. However, applying sampling or not is orthogonal to the graph partitioning problem and is not the focus of this chapter.

Graph Manager

The graph manager maintains the partition structures of the current graph and is responsible for answering queries. The most common query is whether or not a particular vertex is in the current graph, and if so, to which partition it belongs. This question comes from not only the user who would like to know about the graph, but also the partitioning algorithm itself for inserting an edge. Other common queries include (a) for a given vertex, what are the vertices in the same partition? and (b) what is the total number of partitions?

The two key data structures maintained by the graph manager for evolution-aware partitioning are shown in Figure 3.3. The vertex table is a hash table used to maintain the mapping of a vertex ID to a partition ID. The partition table is another hash table used to maintain all the edges in a partition, consisting of the edges forming a connected component of the current graph within the streaming window. The key for the vertex table is the vertex ID and the value is the partition ID. We use the ID of the first vertex in the partition as the partition ID. It is obvious that in this way, two different partitions will never have the same partition ID. The key for the partition table is the partition ID and the value is a pair. The first member of the pair is the partition size, which records the number of vertices in the partition. The second member is a list of edges, storing all the connected edges in this partition. Note a very nice property is that all the incoming edges are naturally sorted by their weights (timestamps). So the only two operations for the list, in a single partition, is adding an edge to the end (where the weight is biggest/the edge is newest) or remove an edge from the front (where
the weight is smallest/the edge is oldest).

![Data structures maintained by the graph manager for evolution-aware partitioning.](Image)

This partition table is efficient for insertion. The tricky part is deletion, because when deleting an edge, we do not know if the rest of the edges still remains as a connected component. While there is an online method to keep track of the connected component [37], the algorithm is quite complicated to implement. We use a very simple method as in [25]: upon a deletion of an edge from a partition, delete the entire partition and then reinsert all the edges except for the deleted one.

In summary, for each insertion, first query the vertex table and find the corresponding partition or create a new partition for the vertices. Then go to the partition table, increase the size of the certain partition and append the edge to the end of its edge list. For each deletion, erase the entire partition and reinsert the rest of the edges.

### 3.3.3 The Algorithm

In this subsection, we present the algorithm. The two main components of the algorithm are insertion and deletion of an edge. We present them separately.

For insertion, we first need to lookup the partition membership of the two endpoints. There are four cases. Case 1: If they are both new, create two new entries in the vertex table and use the ID of the first vertex as the partition ID. Also create a new entry in the partition table and insert the new edge. We do not check the partition size here, because we assume, in reality, the maximum partition size is always great than 2. Case 2: If one of the two endpoints is new, create one new entry in the vertex table and assign it the partition ID of the old endpoint. Also append this edge to the corresponding partition in the partition table. Case 3: If both endpoints are not new and they are in the same partition, simply append this edge...
to the partition. Case 4: Otherwise (if they are not in the same partition),
merge the smaller partition to the bigger one so that the number of entries is
less we need to modify in the tables. The insertion part of the algorithm is
shown in Algorithm 1. Note that we only need to check the constraint for the
maximum partition size in Cases 2 and 4. And in the corresponding parts of
the pseudo-code, we use if (if maximum partition size reached then) clause
rather than while clause, because after each reinsertion, the algorithm will
check the constraint again automatically, if necessary.

Algorithm 1 Evolution-Aware Algorithm: Insertion

\begin{algorithm}
\begin{algorithmic}
  \FOR {each new edge}
    \IF {both endpoints of the edge are new}
      insert them to vertexTable and partitionTable;
    \ELSEIF {only one endpoint of the edge is new}
      insert the new endpoint to vertexTable;
      insert the edge to the partition of the old endpoint;
      \IF {maximum partition size reached}
        delete an edge from this partition;
      \ENDIF
    \ELSE
      \IF {the two endpoints are in the same partition}
        insert the edge to that partition;
      \ELSE
        merge the smaller partition to the bigger one;
        \IF {maximum partition size reached}
          delete an edge from this partition
        \ENDIF
      \ENDIF
    \ENDIF
  \ENDIF
  \IF {window is full}
    delete the oldest edge;
  \ENDIF
\ENDFOR
\end{algorithmic}
\end{algorithm}

For deletion, if it is caused by the violation of the maximum partition size,
we know the partition ID where we need to delete the edge. If it is due
to an expiration from a streaming window, we can have two options. We
could create a min-heap of the partition IDs sorted by the timestamps of
the oldest edge in each partition. So each time we need to delete an edge,
pop the partition ID and go to that partition to delete the edge. Or we
could implement a separate streaming window and use it to indicate which
partition to remove the oldest edge. The deletion part of the algorithm is shown in Algorithm 2. Note in practice, while deleting the partition, instead of erasing all the corresponding entries in the vertex table, one can just set their partition IDs to a value that will never be a legal one (for example, set it to 0 if all the legal partition IDs will be strictly greater than 0) so that we know those vertices are currently not in the graph. The reason for doing so is that deleting/inserting an entry in a hash table could be more expensive than just modifying its value. And we know we will reinsert most of the edges/vertices back immediately.

**Algorithm 2 Evolution-Aware Algorithm: Deletion**

```plaintext
edgeListCopy = the edge list of that partition;
edgeListCopy.pop_front();
erase the entry in partitionTable;
reset the corresponding entries in vertexTable;
for each edge in edgeListCopy do
    insert that edge using the insertion algorithm;
end for
```

3.3.4 Distributed Implementation

Since the graph within the current streaming window could still be too large to fit in a single machine, we can distribute the storage and computational requirements to different hosts in such cases, as showed in Figure 3.4. Due to the lack of space, we only give the outline of the distributed implementation. The vertex table is much smaller than the partition table. So assume it can fit in a single host and we only need to distribute the partition table. Then we can build another hash table so that we can hash each partition ID to a machine ID and store the partition in that machine. Note that for deletion of an edge, all the operations will only happen in that machine. Even if we finally get more than one partition from the splitting, all the resulting partitions can definitely fit in that machine, because their sum size cannot exceed the size of the original partition. But if two partitions in different hosts needed to be merged, we have to move data between the two host machines.

If the vertex table is too big to be hosted in one machine, we can also hash the vertex ID to a machine ID so that the vertex table is distributed in a
similar way.

![Distributed implementation diagram]

Figure 3.4: Distributed implementation.

3.3.5 EAP vs. EIP

Now, we compare qualitatively our EAP scheme with an EIP (evolution-insensitive partitioning) approach [25]. Detailed quantitative comparisons will be presented in Section 3.5. The EIP is a reservoir sample-based algorithm. Similar to our scheme, the EIP is also for partitioning streaming graphs. However, besides relying on the streaming window to capture evolution, the partitioning algorithm itself is not aware of partition evolution. In the EIP, each edge is assigned with a random number. And the system maintains a fraction of the edges whose random numbers are below a sampling threshold. For those sampled edges, they are maintained by the algorithm in a sorted order based on the random numbers. When there is a violation of the MPS constraint, edges are removed one by one from the one with the largest random number. Because of its reliance on the maintenance of edges based the random numbers assigned, unnecessary deletions might happen to partitions whose sizes do not exceed the MPS constraint. These unnecessarily deleted edges will need to be reinserted back to their corresponding partitions. As a result, not only the deletion process takes more steps, but also additional insertions of edges are needed. For more detail about the EIP algorithm, refer to [25].

Figure 3.5 shows the comparisons of time/space complexity. In the best case of insertion, the EAP only needs to insert entries into the hash tables, which costs constant time. In the worst case, an insertion can trigger a merge of two partitions. In this case, it may take $O(M)$, where $M$ is the maximum partition size. But for the EIP, even in the best case, it needs to find a proper position in the reservoir to insert the edge in the sorted order based on the
For edge deletions, the EAP will delete edges only from the correct partition, followed by the reinsertions of the rest of the edges of the partition. This will take $O(M)$. However, for the EIP, it will be a sequence of random deletions until enough of the edges are removed from the right partition. Suppose that each partition is equally likely to be chosen during a deletion. Then it can be viewed as a Bernoulli trial process. The expected number of trials before the first success is equal to the number of partitions in the current graph within the streaming window. After the deletions, reinsertions are also needed. Thus the complexity for the EIP will be $O(n) + O(M)$, where $n$ is the total number partitions.

For memory usage, the EAP needs to maintain a vertex table and a partition table. Since it is implemented using hash tables, it will allocate more memory than actually needed. So we need to consider the load factor of the hash table. This factor is set to be no larger than 0.66 in our experiments. For the EIP, the two reservoirs stores an entire copy of edges. In addition, the graph manager stores the edges and vertices for partitions. Although the graph managers of the EAP and the EIP would store different edges/vertices, the total number of edges/vertices are comparable. As a result, the EAP will be generally more memory efficient than the EIP.
3.4 Evolution Analysis

In streaming graphs, the graph updates are considered as continuous and infinite. The partitioning of vertices in this continuously changing graph could evolve over time. In these scenarios, it would be useful if we could identify the partitioning evolution and even quantify its intensity. In this section, we propose methods for analyzing partitioning evolution.

3.4.1 Evolution of partitions

As proposed in [31], we can understand the changing of the partitions by comparing their structures in two different times. For this comparison, we need two clock times: \( t_1 \) and \( t_2 \) and a window size \( s \). Without loss of generality and for the ease of analysis, suppose \( t_2 > t_1 \) and \( s \) is the time-based window size, for example 24 hours. Let us use the updates arrived between \( t_1 - s \) and \( t_1 \) to construct one graph \( G_{[t_1-s,t_1]} \) and those in \( [t_2-s,t_2] \) to construct another graph \( G_{[t_2-s,t_2]} \). Then the changes happened between \( t_1 \) and \( t_2 \) can be identified by answering the following questions: (1) How many new partitions have appeared and what are they? (2) How many partitions have disappeared and what are they? (3) How many partitions are still there, what are they, and how much have their sizes changed?

It is obvious that EAP is easy for this kind of analysis, because it keeps the table for all the partitions in the current graph. In details, when it is \( t_1 \), output all the partitions and their sizes. When it is \( t_2 \), output again. By comparing the two outputs, we can understand how the evolutions happened between \( t_1 \) and \( t_2 \). It is also pointed out in [31] that when a large fraction of partitions belong to the third case (the partitions retained from \( t_1 \) to \( t_2 \)), it is a sign that the stream is quite stable during this period of time.

It is true that by investigating the partitions presenting at different times can give us useful information about how they evolve. But then in order to understand the general trend in the evolution, we have to sample many time slots, which could be a significant extra overhead. Besides, even all the partitions keep exactly the same from \( t_1 \) to \( t_2 \), there still could be evolutions. For example, the partitions can change a lot during \( [t_1, t_3] \), where \( t_3 < t_2 \), but finally they all mutate back to the original shapes at \( t_2 \). This is quite possible when the streams are periodic. Then if we do not choose the sampling points
carefully, we will overlook these evolutions.

3.4.2 Quantify Stream Stableness

As an alternative, we look into the stream of updates. It is the dynamic updates of the streaming graph that cause the evolution, slowly or dramatically. The changing of the partitions at different clock times are only the effect, i.e. these phenomena are merely the reflection of the evolving streams. Therefore, by monitoring graph updates, we can better understand and capture the partitioning evolution.

**Definition 7** Given a graph and a new edge to be inserted to the graph, this edge is called an unstable edge, if the number of partitions changes after inserting this edge; otherwise, this edge is called a stable edge.

For example, in this chapter, the given graph is the incremental streaming graph within the current window. Hence the definition of the unstableness/stableness follows.

\[
\text{unstablleness} = \text{Ave} \left( \frac{\text{numUnstableEdgesInWindow}}{\text{windowSize}} \right) \quad (3.1)
\]

\[
\text{stableness} = -\log \text{unstablleness} \quad (3.2)
\]

The unstablleness here is measured under a tumbling window. Each time we calculate the fraction of unstable edges among all the edges in one window and at the end, we average them. The bigger the number, the more unstable the stream is. From the experiments we will see soon, it is more convenient to define a corresponding stableness on a log scale. So we have Equation (3.2). The bigger the number, the more stable the stream is.

The motivation of defining such unstablness/stableness is as follows. The unstablleness shows the dynamic property of the graph or stream. The graph is said to be stable when the topology of the graph does not change much. For example, new edges only come into existing partitions so that only the density of the partitions changes. But we are partitioning vertices (that is also why
there is a MPS constraint on the number of vertices per partition), not edges, so the changing in the density is actually “not a change” in our problem. However, when the new edge causes birth of a new partition or merging of partitions, the topology of the graph changes and hence the memberships of the vertices will be changed. Again, we are partitioning vertices, so this is a sign of unstableness for us.

Moreover, the unstableness/stableness is measured without the MPS constraint. In nature, the partitions do not necessarily have such a constraint. It is not obvious that a measurement oracle can be universally designed with the constraint, because there is not a uniform way to solve the MPS violations. Thus, the unstableness/stableness can be viewed as one natural property of the stream.

Last but not the least, the unstableness should not be confounded with the “evolution in the window.” They are related but not the same. If the stream is unstable, there is surely more evolution in the window, for example, frequent merging of partitions causes rapid increasing of partition sizes and frequent changing of vertices memberships. However, evolution in the window also has something to do with the MPS constraint. If there is no such a constraint, then the algorithms will produce the same result. In this case, a sliding window alone can perfectly capture the evolution, because it always keeps the newest edges. But when there is a MPS constraint, we have to decide which edges to be deleted if a violation occurs. And this matters in terms of evolution-aware partitioning, because an algorithm might be blind to the emerging partitions if it does not choose the edge candidates for deletion carefully. In this sense, instead of “capturing the evolution in a window,” an evolution-aware algorithm is actually “maintaining the evolution in a window.” A sliding window help any algorithm to keep the newest edges and hence the emerging partitions. But the algorithm itself should be responsible for carefully solving the MPS constraint violations and maintaining the emerging partitions.

3.5 Experiments

Sets of experiments were conducted to thoroughly evaluate the EAP, including visualizing its ability to capture the emerging evolution, measuring its
quality using the weighted cut size and its efficiency in terms of throughput. All the experiments are run on a single machine with Intel Xeon processor and 126 GB physical memory, which runs Linux Red Hat 4.4 operating system. All the algorithms were implemented in C++.

We compare our algorithm with the EIP and two versions of METIS: the recursive METIS (Rmetis) and the Kway METIS (Kmetis) [24]. We use the METIS software package of Version 5.0 and keep all the default settings, like the maximum allowed load imbalance among the partitions is 1.03. Note in our problem, we specify a constraint on the maximum number of vertices in each partition. However, in METIS, we cannot explicitly specify this constraint. So we use the following approximation: if the total number of vertices is \( V \) and the maximum partition size is \( M \), then the number of partitions METIS will compute is set to \( \lceil V/M \rceil \). As specified in METIS, we use an adjacency list to store the graph. Upon each edge addition or deletion, this list will be modified correspondingly. The weights of the edges are normalized timestamps. Thus partitioning for METIS is just partitioning this weighted graph.

For the data sets, we use one synthetic data set and four real-world ones. The synthetic data is only for visualizing how the algorithms capture the emerging evolution in partitioning. We do not use the real data sets for this purpose, because it is impossible to control their number of vertices/edges or locate precisely the partitioning evolution in the stream for a good visualization. The four real-world data sets are:

1. Twitter: These are twitter replies during a five-day period. They were 1% sampled, so they are sparsest among the four data sets. In this set, vertices denote the twitter users and edges denote the replies. For example, an update/edge \(< v1, v2, t >\) represents that user \( v1 \) replies to \( v2 \) at time \( t \). There are 1.9M vertices and 1.6M edges in total.

2. Web Notre Dame (or Web for short): This data set was obtained by taking a snapshot of the web graph of the University of Notre Dame in 1999. An update \(< v1, v2, t >\) means a web page \( v1 \) points to another page \( v2 \) at time \( t \), i.e. in page \( v1 \), there is a hyperlink pointing to \( v2 \). (We insert random timestamps to this set, because there is no time information recorded originally). There are 330K vertices and 1.5M edges in this data set.
3. Citations: These were crawled from the paper citations of high energy physics in arXiv. They cover a time range of 124 months. An update $< v1, v2, t >$ in this setting means that paper $v1$ cites paper $v2$ at time $t$, which is the actual publication date of $v1$. There are 34K total vertices and 350K edges.

4. DNS request (or DNS for short): These are the DNS requests from the computers within the IBM Watson Research Center to domains on the web in a day. Thus, an edge $< v1, v2, t >$ means computer $v1$ makes a DNS request of domain $v2$ at time $t$. We observe that some machine requests the same IP many times a day. This implies the graph could be very dense. There are 180K vertices and 4.8M edges in it. Thus the average degree of a vertex is almost 27. This is a sign of a dense graph.

We design three kinds of experiments:

- Visualization of evolution: We plot the real partition structures of the graph by taking snapshots during the execution of the partitioning algorithms. We only do this for the synthetic data set because we can then control the number of vertices/edges, the evolution strength, and the evolution starting/ending time for best visualization.

- Quality experiments: For the quality metric, we measure the weighted cut size of the partitioning algorithm, which reflects how well an algorithm can capture the evolution. Note that it is difficult to measure the “real-time cut size” of the partitioning because the interpartition edges are not kept by the partitioning algorithms. Instead, we design a quality experiment based on a tumbling window. For a tumbling window of size $s$, we run the partitioning algorithm until $s$ edge updates arrive. Then we use the partition structures, e.g., the partition table in the EAP, obtained up to that moment for measuring the weighted cut size by re-scanning all the edges from the tumbling window again. For each edge, if both its endpoints are in the same partition, we skip this edge; otherwise, we accumulate its weight toward the weighted cut size. We repeat this for the rest of the data set and compute the average of the weighted cut sizes associated with all the tumbling windows at the end.
• Throughput experiments: For the efficiency metric, we run the algorithms and calculate how many updates they can handle in one second.

3.5.1 Capture the Evolution

In this part, we first use a synthetic data set to show how well our EAP scheme captures the emerging partitioning evolution, comparing to the EIP approach [25]. Then we show the performance of the algorithms using four real-world data sets. We show their differences under different stableness measures as defined in Section 3.4.

To construct the synthetic data set for this experiment, we first create two partitions of vertices, each with a total number of 200 vertices, i.e. vertices with ID 1-200 for the first partition and 201-400 for the second. Then we randomly create 400 base edges for each of the two partitions, respectively. After the 800th edge, we start to create interpartition edges. We use an interpartition edge probability $p_{ic}$ to control the intensity of partition evolution. For each incoming edge, with probability $p_{ic}$, it connects two vertices in different partitions, and with $1 - p_{ic}$, two vertices in the same partition. In order to get a better visualization, an interpartition edge only chooses a vertex from the right half of the first partition (vertex ID 101-200) and the left half from the second partition (vertex ID 201-300).

Figure 3.6 shows three snapshots of the partitioning structures for each algorithm with METIS on the top, EAP in the middle, and EIP on the bottom. There are three columns: on the left are partitioning snapshots for the initial graphs (after inserting the 800 base edges), in the middle are the partitioning snapshots after 150 additional updates, and on the right are the partitioning snapshots after 300 additional updates. Each partitioning snapshot is plotted on a unit cartesian coordinate system, although we do not show the $x/y$ labels for brevity. The position of each vertex in the figure is determined as follows: for each vertex, we set its $x$ coordinate to be $\text{vertexID}/400$ and its $y$ coordinate to be a random number uniformly sampled from $[0, 1]$. We set $p_{ic}$ to 0.9 and the maximum partition size to 200. The red circles, blue diamonds, and black squares represent different partitions. Note that we use different colored shapes to represent partitions and do not show edges in Figure 3.6 as they would have made the visualization
of the partitioning difficult.

With this synthetic workload, a new partition should be forming in the middle of the figure, as shown in the first row of Figure 3.6, which is the result from METIS and we regard this as the optimum. It is clear to see in Figure 3.6, after 300 updates, the EAP identifies clearly the new partition shown in black squares. Namely, it captures the emerging evolution and follows the emerging trend in the stream. However, the EIP does not capture this change, i.e., the two initial partitions are mixing their members and the new partition is still not emerging. Also, the partitions produced by METIS are denser than that by EAP, because EAP will delete edges when solving MPS constraint violations. Note in plotting the second and third snapshots of the EAP, the red circles or the blue diamonds may have more than one partition within them. However, all the black squares are in the same partition. For the second and third snapshots of the EIP, the red circles and blue diamonds are single partitions, respectively. But the black squares have multiple partitions. Nevertheless, for all the snapshots, if two vertices have different colors/shapes, they are definitely in two different partitions.

![Figure 3.6: Evolution of partitioning. The red circles and blue diamonds are the two original partitions. The black squares are the new emerging partition. EAP is able to capture the evolution after 300 updates, while EIP not.](image)

Figure 3.7 shows the weighted cut sizes of the EAP and EIP for this synthetic data set with different interpartition edge probabilities and different maximum partition sizes. EAP is generally better than EIP in the weighted cut size (the smaller the better). This is because EAP always removes the oldest edge (edge with least weight) first in order to capture the emerging evolution, while EIP removes edges randomly. As the interpartition probability decreases from 0.9 to 0.1, the weighted cut size will first increases...
and then decreases for both algorithms. This is because when the trend of forming the new partition is uncertain, i.e. $p_{ic} = 0.5$, it becomes a “random cut” for both algorithms, which results in the largest weighted cut sizes for both algorithms. When $p_{ic}$ is larger than 0.5, the trend is more and more obvious, and hence the weighted cut size begins to drop. When it is below 0.5, the two original partitions will be more and more stable, so the weighted cut size will also drop for both EAP and EIP. Another way to understand the shapes of the curves is to consider the two extreme cases when $p_{ic} = 0$ or 1. When $p_{ic} = 0$, it means that there is no change in partitioning, so the weighted cut size will be 0. When $p_{ic} = 1$, it means that after enough updates, all the vertices will come to the middle and hence the cut size will also be 0 finally. Notice that when $p < 0.2$, EIP is a little better than EAP, because EAP might have over-reacted to the changes. But this difference is relatively small.

For the figure on the right of Figure 3.7, when the maximum partition size, $M$, is bigger, the weighted cut size is smaller, because now each partition can hold more vertices, leading to less interpartition edges. For all the cases using this data set, EAP yields a lower weighted cut size, which is better. This shows EAP captures the partitioning evolution better.

Figure 3.7: Quality experiments for the synthetic data. EAP is generally better than EIP.

Before studying the difference of performance for the two algorithms under different real-world streams/data sets, let us first show the characteristics of the four real-world data sets. Figure 3.8 shows their stableness under different window sizes. From this figure, Twitter is the most unstable, while DNS the most stable. We also show the standard deviation using the error bars (the little short line segments on the top of each bar). They are small, so is the deviation. Notice for a particular stream, the bigger the window, the “stabler” the stream. In fact, this is incurred by the artificial tumbling
window. For example: for a given stream of \( n \) edges, we could use two methods to measure it: a window of infinite size and a tumbling window of size \( n/2 \). Suppose (1) using the infinite window, we get \( k \) unstable edges; and (2) using the tumbling window of size \( n/2 \), we get \( k_1 \) and \( k_2 \) unstable edges respectively in the two snapshots. Then (1) \( \text{unstability}_1 = k/n \); and (2) \( \text{unstability}_2 = (k_1/(n/2) + k_2/(n/2))/2 = (k_1 + k_2)/n \). And it is not hard to see \( k >= k_1 + k_2 \) definitely according to the definition of an unstable edge. Thus a bigger window will “produce stabler streams.” This is a little misleading because we say the \textit{stability} is a natural property of a stream, so we should not be able to change it using a window. Although this discrepancy can be fixed by measuring the stability under an infinite window, we still present the results under finite windows because all our quality/throughput experiments are conducted such ways and we want to know the stability in these circumstances.

![Figure 3.8](image_url)

**Figure 3.8:** Different properties of the data sets: Twitter is the most unstable, while DNS the most stable.

Figure 3.9 shows the performance of the two algorithms in terms of ratios of the weighted cut sizes and throughputs under different streams. On the \( x \)-axis, we plot the stability measures of the different data sets. The two dotted lines are comparisons of the ratios of weighted cut sizes. The red line with square markers represents experiments when the maximum partition size is set to 1K, while the black circle-marked line represents experiments when the maximum partition size is set to 2K. Because EAP generally yields a smaller weighted cut size, we use the ratio of the weighted cut size of EIP divided by that of EAP to see their differences. When the stream is less stable, the difference is bigger: EAP can be about six times better than EIP under the Twitter stream with \( M \) equaling to 1K. When the stream is very stable, the difference is smaller. So the ratio is close to 1 at that point.
The two solid lines in Figure 3.9 compare their throughputs. The blue line marked by diamonds shows the throughput ratios under four different streams with MPS of 1K, while the magenta one with triangle markers is that with MPS of 2K. Since EAP can process more insertions each second, we use the ratio with the throughput of EAP divided by that of EIP. When the stream is less stable, EAP can be up to eight times better than EIP in both cases. Even when the stream is quite stable, EAP can still produce about three times better throughput. This is because the simpler architecture and the efficient data structure of EAP makes the algorithm run much faster.

![Graph showing throughput comparison between EAP and EIP with different MPSs]

**Figure 3.9:** EAP vs EIP with different stableness. EAP is more advantageous when the stream is less stable

### 3.5.2 Experiments with Different MPSs

Here, we show quality and throughput experiments with different maximum partition sizes. Quality is measured by averaging the weighted cut sizes for all the tumbling windows of the data sets. It is important because it reflects to a large degree how well an algorithm captures the partitioning evolution. Throughput reflects the efficiency of the algorithm. In the following experiments, we also use METIS for comparison. It is considered as the best offline algorithm in terms of quality. So it can serve as an optimum an algorithm can reach.

Figure 3.10 shows the quality experiments with different $M$’s. METIS is the best, but EAP is generally better than EIP. We can see when $M$ increases, the weighted cut size decreases accordingly, because now each partition allows more vertices, resulting in fewer interpartition edges. We also notice that the
weighted cut size produced by METIS is not that sensitive to $M$ for some data sets, such as the Citations and the DNS Request data sets.

Figure 3.10: Quality experiments with different maximum partition sizes. METIS is the best and EAP is better than EIP.

In order to show more clearly the relative performance of the various algorithms, we also plot a bar chart in Figure 3.11. The weighted cut size is normalized by that of Kmetis. Once again, we see that EAP is generally better compared to EIP, especially when the stream is more dynamic. EAP is also closer to METIS in such an environment, especially if we look at its performance under the Twitter data set. An interesting thing one can notice is that in the Twitter figure, when $M$ is 2K, EAP is actually a little better than METIS. Recall that we have to use an approximation for METIS for conducting the quality experiments, because we cannot set the exact maximum partition size constraint for METIS. This creates negative impacts on METIS in some cases, because it is always trying to do a balanced partitioning. Also notice that in some cases, EIP is better than EAP, like in the DNS figure when $M$ is less than 4K. This is because when the graph is rather stable, EAP may overreact to the change in the stream, resulting in removing unnecessary edges.

Figure 3.12 shows the throughput of the algorithms with different $M$'s. We already saw that EAP will be more advantageous when the stream is less stable. In this set of figures, EAP is the best in all the cases and EIP
is the second. METIS is slow because it is an offline algorithm and is not able to do incrementally partitioning upon each insertion of an edge. When the maximum partition size is bigger, the throughput will generally decrease at first for both EAP and EIP, for example as shown in the Citations figure when $M$ increases from 1K to 5K. Generally, a bigger $M$ means a bigger average partition size. Thus, when there is a deletion, the number of edges to be removed/reinserted will be larger (note the deletion is expensive for both EAP and EIP). However, the throughput will not continue to decrease as $M$ keeps on growing. This is because there will be fewer violations of this constraint as $M$ becomes large, reducing the chance that we need to do the expensive deletion operations. As a result, after some turning point, the throughput tends to scale with $M$. At that time, the deletion operation will be mainly caused by an expired edge from the streaming window, which will happen rather stably and hence make the throughput stable too. However METIS is less sensitive to $M$, because the running time of METIS depends on the number of vertices, rather than the number of edges.

We also observe that if the graph is more sparse, the throughput will be higher for both EAP and EIP. For example, they will have throughput of the order of $10^5$ when processing the Twitter data. But for the DNS data set, which is densest among the four data sets, the throughput will drop to the order of $10^2$. When the graph is sparse, the structure is simpler. For example, the average number of vertices/edges in one partition is smaller.
and hence the average cost for each deletion is lower. Moreover, for a sparse graph and a dense graph, if we set the same $M$ for them, there tends to be a smaller chance for the sparse graph to violate the MPS constraint. And if it occurs, the number of edges to be removed to solve the constraint is less for a sparse graph. Both reduce the running time of the algorithm and increase the throughput. That is in part why we set different sets of $M$’s in this experiment for different data sets.

3.5.3 Experiments with Different Window Sizes

Figure 3.13 shows the quality experiments with different window sizes. If the window size increases, on one hand, the weighted cut size for each algorithm will increase. A bigger window implies more vertices/edges in the current graph. Then if $M$ is still the same, there tends to be more interpartition edges generally, hence causing the bigger weighted cut size. On the other hand, when the window size increases, the relative performance of different algorithms will be more significant. This shows that even though streaming window can naturally help any streaming partitioning algorithm capture some partition evolution, it alone may be insufficient, especially when the window size is large and the partitioning of the graph is unstable. For example, for the Web data set, when the window size is 40K, all the algorithms produce similar quality. That means, when the window is small,
the sliding window itself does the job of capturing evolution for all the algorithms, since there are not many vertices within the window and the MPS constraint is rarely violated. But when the window size is 100K, METIS can be almost 10 times better than EIP, since now there is a lot of evolution happening within the window and each algorithm is responsible for capturing the evolution now. In the citations and DNS figures, EAP and EIP are close, because these two graphs are more stable and the advantage of EAP is not that obvious.

Moreover, when the stream is relatively more stable, METIS is always the best. But for the figure of the Twitter data, when the window size is larger than 30K, EAP begins to be better than Rmetis. An unstable stream tends to cause more partitioning evolution, and therefore, in this case, the advantage of EAP is more significant than in the other three cases.

Figure 3.13: Quality experiments with different window sizes. When the window is larger, the difference between MEITS (the optimum) and EAP/EIP is bigger, which shows that streaming window alone is not adequate to capture the partitioning evolution.

Figure 3.14 shows the throughput regarding to window size up to 40K. Note that this is when all the algorithms are similar in terms of quality measurement. But even in those cases, EAP is still consistently the best, followed by EIP, and METIS generates little throughput compared to them. The throughput of EAP also exhibits this advantage when the window size is bigger. Note that we do not show the throughput of a large window in the figure, in order to highlight the performance of EAP in small windows.
when they cannot be differentiated by quality measurement. For large windows, please refer to the throughput tests in Figure 3.9. The throughput of METIS has been insensitive to $M$. However, by looking at the two figures corresponding to the Citations and DNS data sets, we can notice that there is now a slope for the METIS lines: if the window is smaller, the throughput is higher. A smaller window means a fewer number of vertices and hence METIS can run faster. Recall that the running time of METIS depends on the number of vertices it needs to handle.

![Throughput experiments with different window sizes.](image)

Figure 3.14: Throughput experiments with different window sizes. EAP is the best.

### 3.6 Summary

In this chapter, we proposed an evolution-aware algorithm for partitioning streaming graphs. We designed an efficient data structure to implement the algorithm. We visualized the process of how EAP can capture the evolution. We designed an evolution analysis scheme to quantify the stableness of the streams. In general, unstable streams tend to cause more partition evolution. We conducted a set of quality experiments to show how well an algorithm can capture the partitioning evolution. Among all the algorithms, METIS is the best. But EAP is better than EIP and the quality is reasonably
good, compared to METIS. We also conducted another set of throughput experiments to show the efficiency of the algorithms. In this comparison, EAP is the best and it generates orders of magnitude higher throughput. Last but not least, we also observe that if the stream is more unstable, i.e. there is more evolutions happening within it, the advantage of EAP is more obvious than EIC, in both quality and throughput.
In this thesis, we propose two methods for weighting the edges for dynamic partitioning of online social networks.

In Chapter 2, we develop algorithms for partitioning access frequency weighted graphs. We pointed out that the objective of partitioning is to keep the two-hop neighborhood of a user in one partition, instead of the one-hop network usually considered. Two-hop neighborhoods are the basic units of retrieval in OSN and can be much larger than one-hop networks. We propose to partition not only the spatial network of social relations, but also in the time dimension so that users who have communicated in a given period are grouped together. We build an activity prediction graph to keep in one partition newly created data that is highly likely to be accessed together. We use a static partitioning method based on KMetis, and a dynamic local partitioning method that requires only a small amount of data movement across partitions. The partitioning results are tested with emulation of Facebook page downloads, and show that the static algorithm achieves 5.6 times better data locality than hash-based partitioning, and the dynamic algorithm achieves 6.4 times better locality while keeping the number of movements small. Almost all queries are kept in at most three partitions for both algorithms.

In Chapter 3, instead of considering more about the access frequency, we are interested in maintaining the newest status of the network and hence the graph is weighted by recency. We propose to view the evolving network as a streaming of graphs sampled at different time. The graph is incremental and the updates to the graphs are given as a stream of edge additions and deletions. Then we present an evolution-aware algorithm for partitioning these streaming graphs dynamically. We incrementally manage individual connected components as partitions. For each partition, we keep the relative recency of edges in sorted order and favor more recent edges for partition
merging and splitting. We evaluate the effectiveness of EAP and compare it with a previous state-of-the-art evolution-insensitive partitioning algorithm. The results show that EAP is both effective and efficient in capturing evolution, achieving up to six times better than EIP in partitioning quality and up to eight times better in throughput based on a set of real-world graphs.

In practice, which weighting method is chosen will depend on the real application. Consider the two cases mentioned in the Introduction: the efficient distributed storage of social networks and the real-time targeted recommendation system. Clearly, the first one should choose weights based on access frequency, since the objective is to reduce the frequency of remote retrieval of data. And the second one should choose weights based on recency, since we always need the newest status of the network. We have already seen encoding the past graph information into the edge weights is a good way for achieving better partitioning and the corresponding benefits. In general, we need to design the weights according to the actual problem.

Future work includes designing an algorithm that takes into consideration a larger portion of the network, instead of focusing on single nodes, so as to further improve partitioning quality while reducing computation complexity. Another angle is to design smart edge weights which can combine multiple partitioning objectives so that we can achieve multiple benefits at the same time.
REFERENCES


